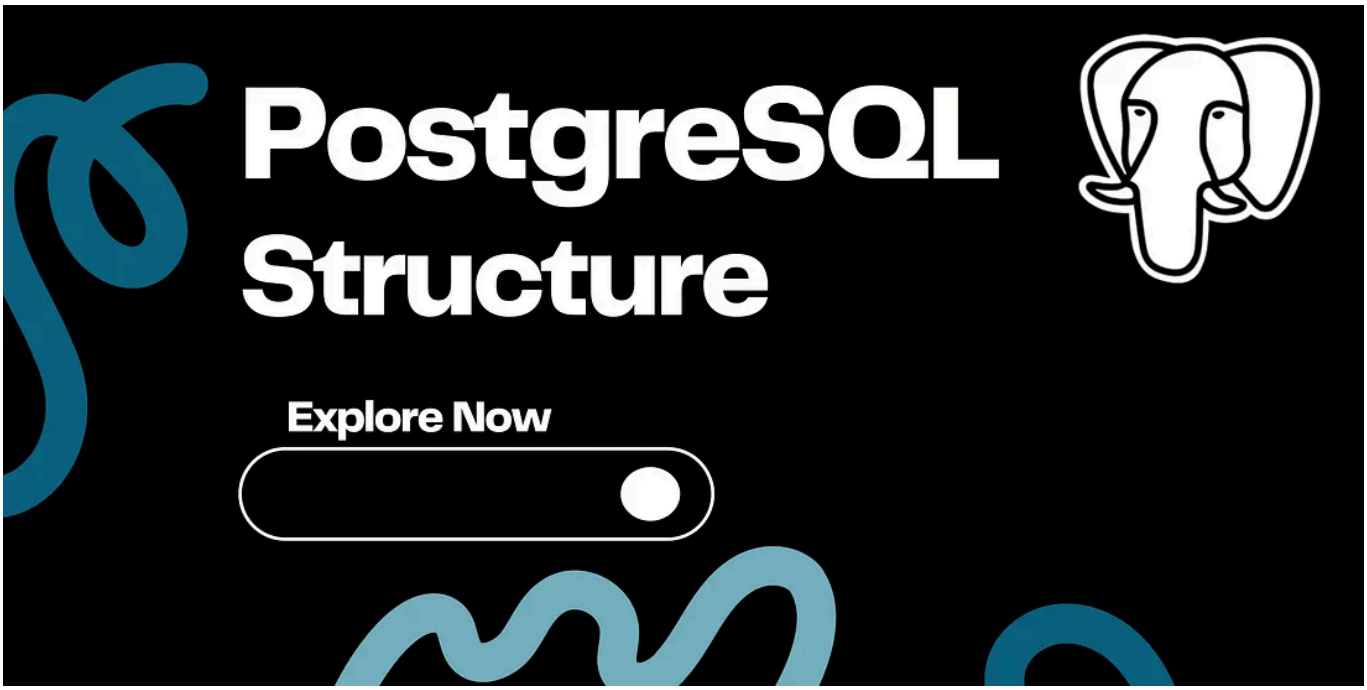
PostgreSQL Architecture

Bitnine Global · Follow
Published in Bitnine Global · 7 min read · Apr 10, 2024

52



Let's dive into the world of PostgreSQL Architecture — or Postgres, as it's fondly known. At first glance, its architecture might seem pretty straightforward, but there's a lot of magic happening under the hood. It's a blend of shared memory, a handful of background processes, and data files. (*See **Figure 1–1***).

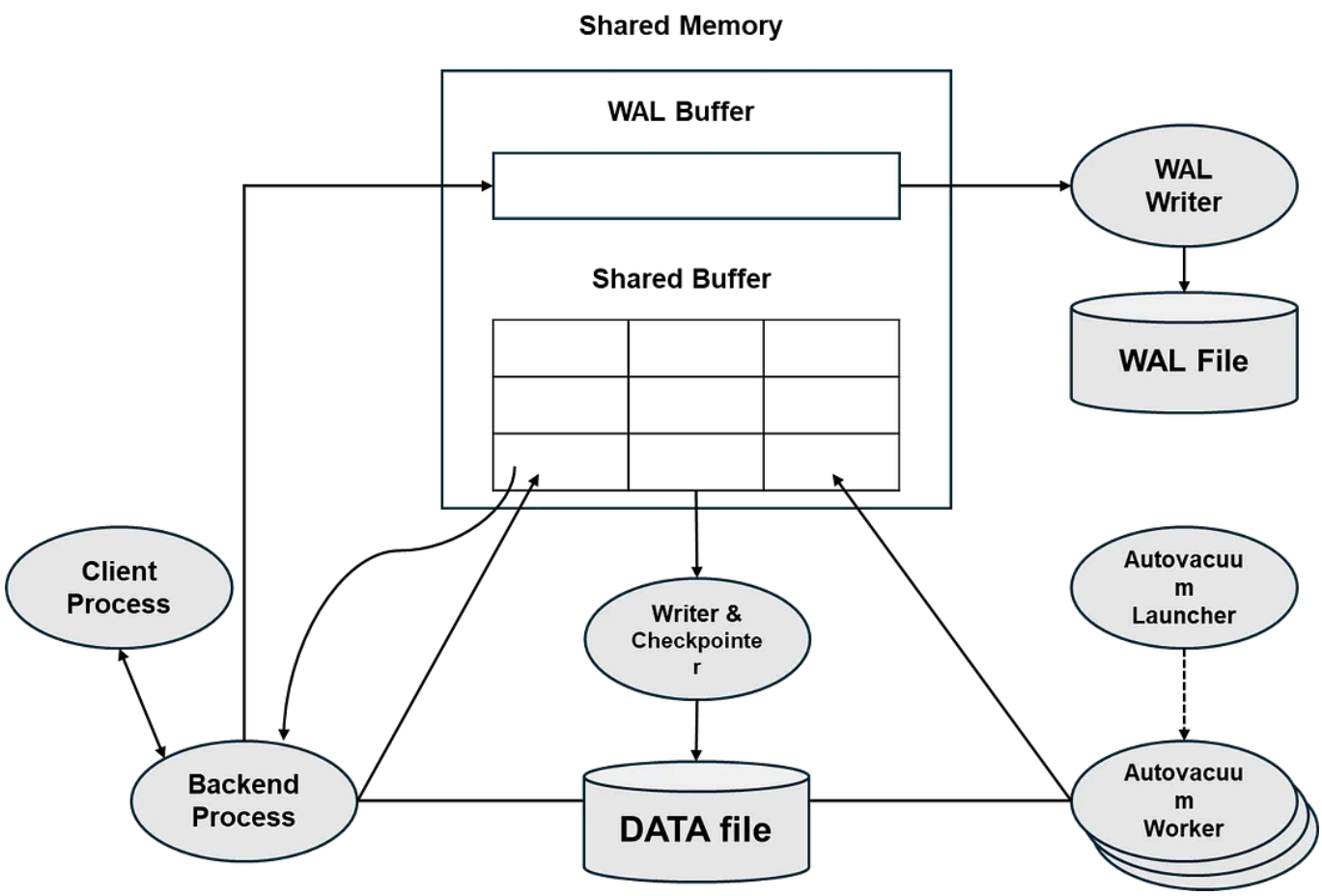


Figure 1–1. PostgreSQL (Postgres) Architecture

1–1. Shared Memory

At the heart of shared memory lies the Shared Buffer and the WAL Buffer. Utilizing tools like pgAdmin, administrators can gain insights into the utilization and performance of these buffers, aiding in the efficient management of PostgreSQL databases. Here we will explain a little more of each of them:

Shared Buffer

The purpose of the Shared Buffer is to minimize disk I/O, as all databases do. To do this, it must meet the following criteria:

- Very large (tens or hundreds of GB) buffers need to be accessed quickly
- Minimize contention when accessed by many users simultaneously
- Frequently used blocks should stay in the buffer as long as possible

WAL Buffer

The WAL Buffer is a buffer that temporarily stores changes to the database. The content stored in the WAL Buffer is written to a WAL file at a fixed point in time. From a backup and recovery perspective, the WAL Buffer and WAL files are very important. (Like Oracle's Redo Log Buffer and Redo Log File).

If you use pgAdmin, you can use it to monitor the WAL Buffer's activity, including the frequency and volume of data being written. This visibility is good for understanding the impact of the WAL Buffer on overall database performance and for making decisions about backup and recovery strategies.

1–2. Process Types

Postgres plays host to four main types of processes:

- Postmaster (Daemon) Process
- Background Process
- Backend Process
- Client Process

1–3. Postmaster Process

The Postmaster Process is like the welcoming committee when you boot up Postgres. It gets the ball rolling with recovery operations, warming up shared memory, and waking up background processes. Plus, it's the one calling the shots by creating backend processes for client connections.

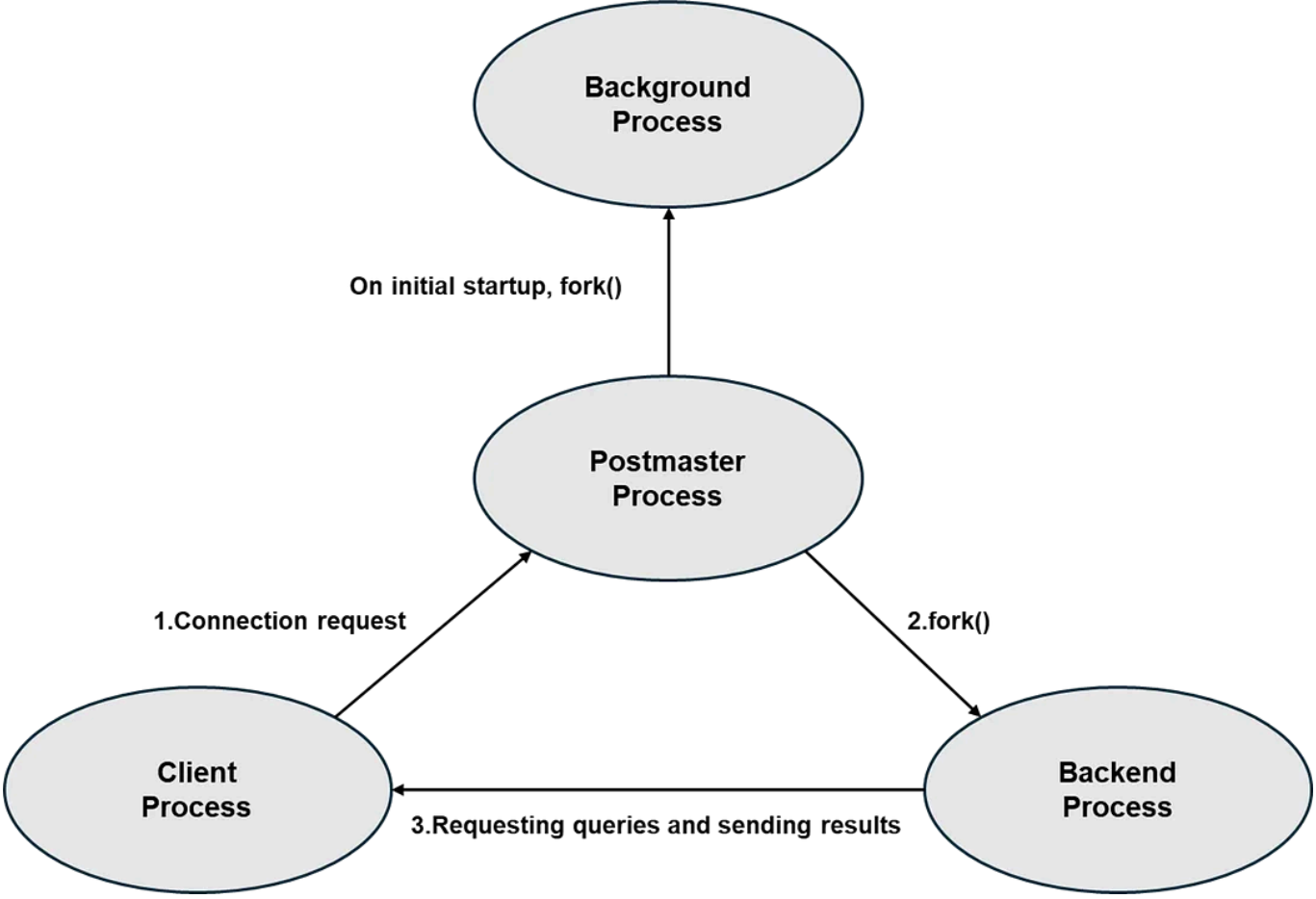


Figure 1–2. Process relationship diagram

If we check the relationship between the processes with the pstree command, we can see that the Postmaster process is the parent process of all the processes.

**For clarity, we've added the process name and argument after the process ID*

```
[postgres@localhost pg11]$ pstree -p 20936
postgres(20936) /home/postgres/pg11/bin/postgres -D /data/pg11
├─postgres(20937) postgres: logger
├─postgres(20939) postgres: checkpointer
├─postgres(20940) postgres: background writer
├─postgres(20941) postgres: walwriter
├─postgres(20942) postgres: autovacuum launcher
├─postgres(20943) postgres: stats collector
└─postgres(20944) postgres: logical replication launcher
```

1–4. Background Process

The essential background processes for PostgreSQL's operation are outlined below. With the exception of the Autovacuum Launcher, these processes are commonly encountered within *ORACLE*, underscoring their foundational role in database management.

Process Name	Function
logger	Logs error messages and a variety of information to log files
checkpointer	Writes dirty buffers to disk when a checkpoint occurs
writer	Periodically writes dirty buffers to disk
wal writer	Writes contents of the WAL Buffer to WAL files
autovacuum launcher	Fork the autovacuum worker when vacuum is required
archiver	When in Archive log mode, copies the WAL file to the specified directory
stats collector	Collect DBMS usage statistics information such as session activity information (pg_stat_activity) and table usage statistics information (pg_stat_all_tables)

1–5. Backend Process

The max number of backend processes is decided by the `**max_connections**` parameter, typically set to 100. These processes are the muscle, doing the heavy lifting by executing queries and passing results back to users. They rely on a bunch of memory structures, affectionately known as local memory, to get the job done.

The main parameters related to local memory are as follows:

work_mem parameter

This is the space used for sort operations, Bitmap operations, Hash join and Merge join operations. The default setting is 4 MB.

maintenance_work_mem parameter

Space used for Vacuum and Create Index operations. The default value is 64 MB.

temp_buffers parameter

Space for storing temporary tables. The default value is 8 MB.

Open in app ↗ Sign up Sign in

Medium Search Write

Database Essentials

1. PostgreSQL, often abbreviated as Postgres, consists of multiple databases, collectively known as a **database cluster**
2. Upon executing `**initdb**`, three databases are created: `**template0**`, `**template1**`, and `**postgres**`.
3. `**template0**` and `**template1**` serve as template databases for creating user databases and include system catalog tables
4. Right after `**initdb**` is run, both `**template0**` and `**template1**` have identical table lists. However, template1 allows for the creation of objects

necessary for users

5. User databases are created by duplicating the **`template1`** database, simplifying the process of creating specific objects in each user database

Tablespace Details

1. Immediately after executing **`initdb`**, two tablespaces are created: **`pg_default`** and **`pg_global`**
2. If a tablespace is not specified when creating a table, it is stored in the **`pg_default tablespace`**
3. Tables managed at the database cluster level are stored in the **`pg_global`** tablespace
4. The physical location of **pg_default** is **`$PGDATA\base`**
5. The physical location of **pg_global** is **`$PGDATA\global`**
6. A single tablespace can be utilized by multiple databases. Within the tablespace directory, subdirectories are created for each database, named after the database's OID
7. When a user tablespace is created, a symbolic link related to the user tablespace is generated in the **`$PGDATA\tblspc`** directory

Table-Specific Information

1. Each table is associated with three files
2. One for storing table data, named after the table's OID
3. One for managing the table's free space, named **`OID_fsm`**
4. One for managing the visibility of table blocks, named **`OID_vm`**
5. Indexes lack a vm file, thus consisting of only two files: **`OID`** and **`OID_fsm`**

> ***Note:*** *At the point of table and index creation, the file name is the OID, and at this moment, the OID and* ***`pg_class.relfilenode`*** *values match. However, after operations like Truncate, Cluster, Vacuum Full, or Reindex, the affected object's*

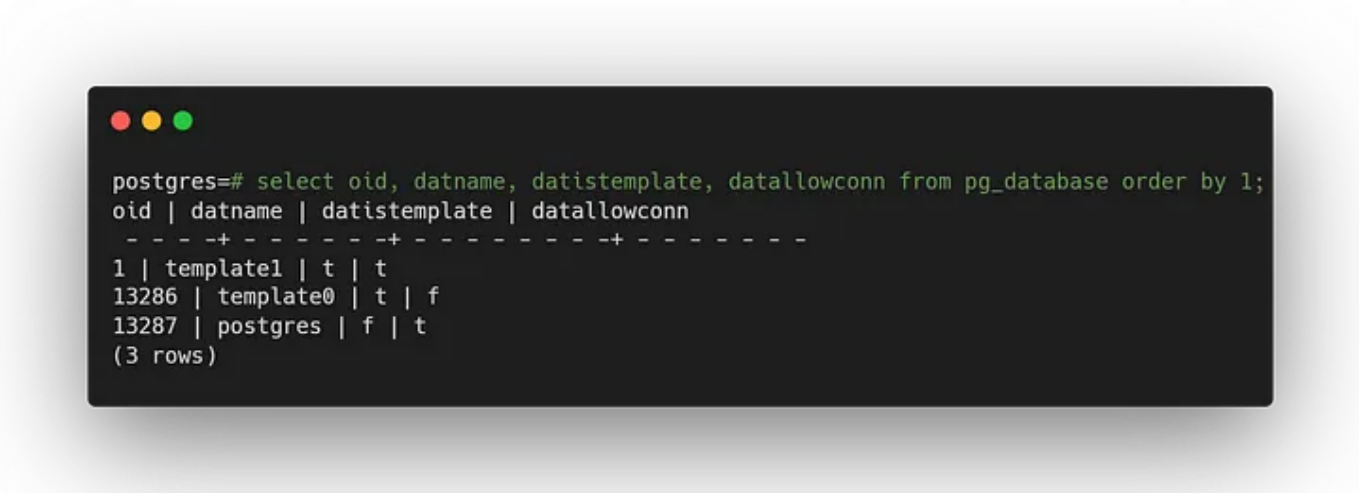
> *`**relfilenode**` value changes, and so does the file name, to match the `**relfilenode**` value.*

For convenience, the `**pg_relation_filepath('object_name')**` function can easily reveal the file's location and name.

`template0`, `template1`, and postgres databases

Let's verify the previously discussed points through testing!

After running `**initdb**`, querying `**pg_database view**` confirms the creation of `**template0**`, `**template1**`, and `**postgres**` databases.


postgres=# select oid, datname, datistemplate, datallowconn from pg_database order by 1;
oid | datname | datistemplate | datallowconn
1 | template1 | t | t
13286 | template0 | t | f
13287 | postgres | f | t
(3 rows)

- **datistemplate Column:** This indicates whether the database is a template for creating user databases. Both `**template0**` and `**template1**` are template databases designed for this purpose.
- **datallowconn Column:** It signifies the database's accessibility for connections. The `**template0**` database cannot be accessed, thus its contents cannot be altered.
- **Purpose of Two Template Databases:** `**template0**` is provided as an initial state template, while `**template1**` allows users to add custom templates. This facilitates user-specific customizations right from database creation.
- **The postgres Database:** By default, this is the primary database created using the `**template1**` database. If no specific database is mentioned

upon connection, it defaults to the postgres database.

- **Database Location:** All databases reside under the `**$PGDATA/base**` directory. The directory name corresponds to the database's OID number.

[postgres@localhost base]$ ls -l
total 36
drwx - - - . 2 postgres postgres 8192 Dec 11 06:04 1
drwx - - - . 2 postgres postgres 8192 Dec 11 06:04 13286
drwx - - - . 2 postgres postgres 8192 Dec 11 06:06 13287

Creating User Databases

User databases are generated by cloning the `**template1**` database. To illustrate this, we first create a sample table T1 within the `**template1**` database. Next, we proceed to create a new database named `**mydb01**` and verify the presence of the T1 table.

```
- - Connect to the template database and create the T1 table
template1=# create table t1 (c1 int);
CREATE TABLE
•
 - mydb01 - Create the mydb01 database
postgres=# create database mydb01;
CREATE DATABASE
•
 - - Connect to the mydb01 database and verify that the T1 table exists
mydb01=# \\dt
List of relations
Schema | Name | Type | Owner
 - - - - + - - - - + - - - - -+ - - - - - -
public | t1 | table | postgres
(1 row)
```

Illustrating this process graphically yields the following depiction:

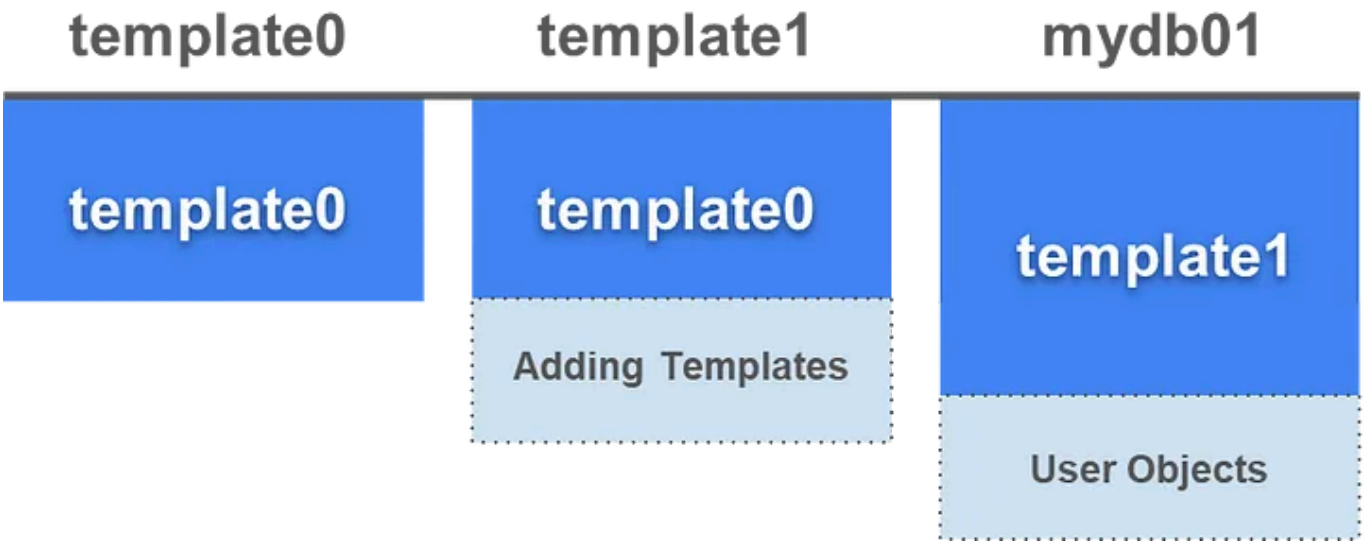


Figure 1–3 Relationship between the template database and the user database

The pg_default Tablespace

After running initdb(), checking the pg_tablespace will reveal the creation of both the `**pg_default**` and `**pg_global tablespaces**`.

```
postgres=# select oid, * from pg_tablespace;
oid | spcname | spcowner | spcacl | spcoptions
 - - - + - - - - - - - + - - - - - + - - - - + - - - - - - -
1663 | pg_default | 10 | |
1664 | pg_global | 10 | |
(2 rows)
```

The pg_default tablespace is located at `**$PGDATA\base**`. This directory contains subdirectories for each database OID, representing the physical structure where each database directory exists under its respective tablespace. *(Refer to Figure 1–4)*

```
[postgres@localhost base]$ ls -l
total 48
drwx - - - . 2 postgres postgres 8192 Dec 11 06:17 1
drwx - - - . 2 postgres postgres 8192 Dec 11 06:04 13286
drwx - - - . 2 postgres postgres 8192 Dec 11 06:06 13287
drwx - - - . 2 postgres postgres 8192 Dec 11 06:19 16387
•
 - Check the OIDs of the databases
postgres=# select oid, datname from pg_database order by 1;
oid | datname
 - - - -+ - - - - - - -
1 | template1
13287 | postgres
16387 | mydb01
(4 rows)
```

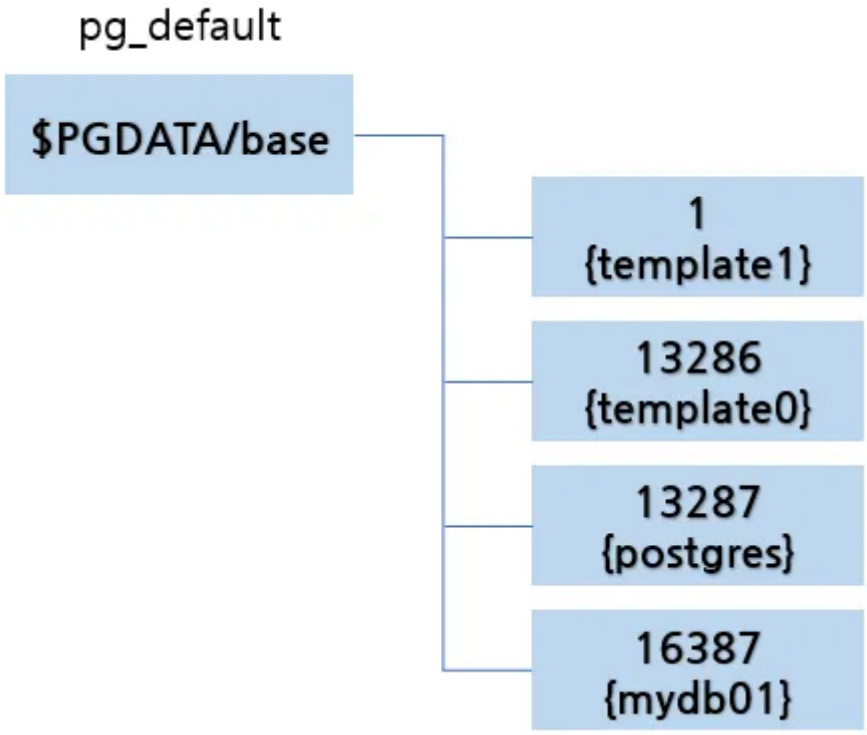


Figure 1–4. Relationship of the pg_default tablespace to the database from a physical configuration perspective

The pg_global Tablespace

The `**pg_global tablespace**` serves to store data that needs to be managed at the **'database cluster'** level.

- For instance, tables like pg_database provide the same information regardless of which database accesses them, underscoring their cluster-wide relevance (Refer to Figure 1–5)
- The location of the `**pg_global**` tablespace is at `**$PGDATA\global**`

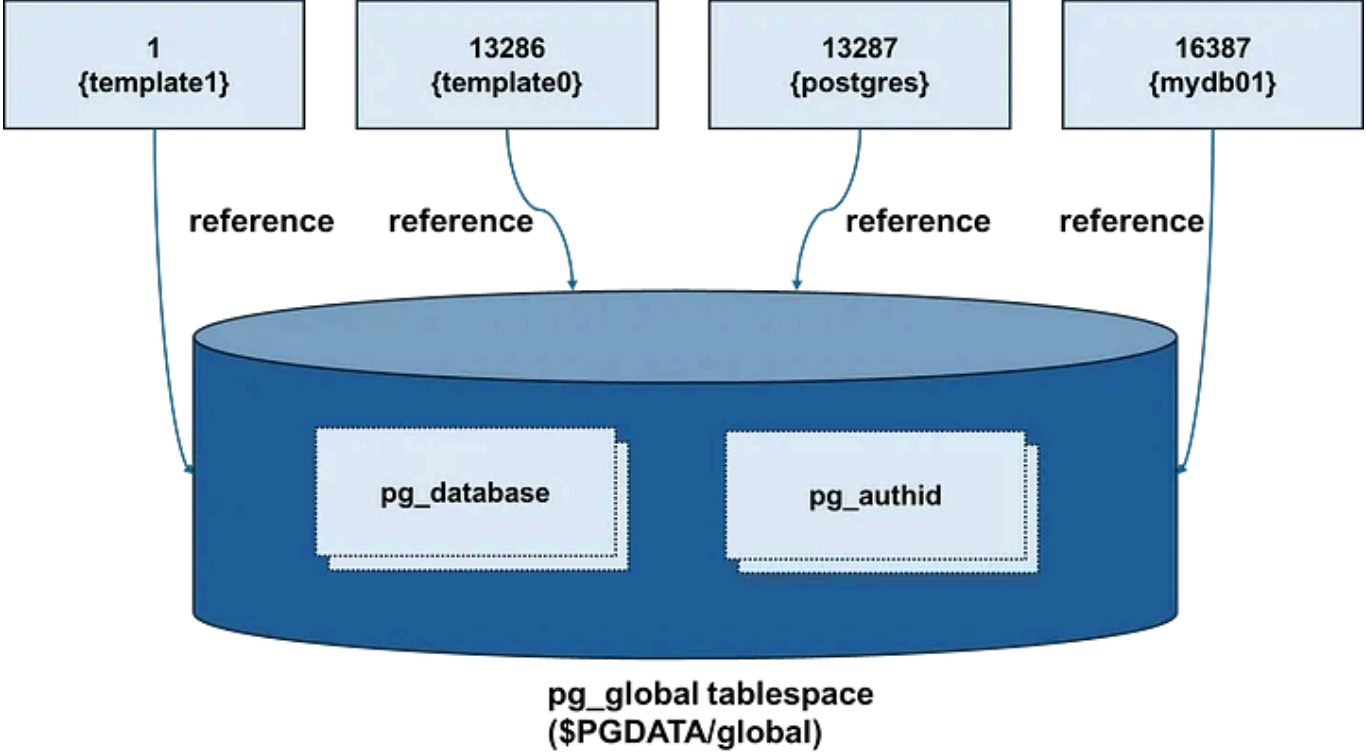


Figure 1–5. Relationship between the pg_global tablespace and the database

Creating User Tablespace

Let's dive into how changes unfold after creating a user tablespace.

```
-- user create
create role hr password hr;

-- On the physical server, run the directory create
sudo mkdir -p ./tablespaces/hr
sudo chown -R postgres ./tablespaces/hr

-- create tablespace with owner as hr
create tablespace hr owner hr location '/data/tablespaces/hr';

-- Create a table in that tablespace
create table test ( col1 text, col2 text) tablespace hr;
```

In PostgreSQL, the entire directory specified by the `**PGDATA**` environment variable is utilized as the database. Subsequently, tables within this database are created as files. This implies that even without explicitly creating a separate tablespace, user tables can be established within the database. Essentially, the directory designated as the database is recognized as a basic tablespace.

According to PostgreSQL's official site, tablespaces are defined as file system paths where database objects can be stored, as determined by the database administrator.

Once a tablespace is created, it can be referred by name when creating database objects.

The capability to create tablespaces, as enabled by the disk layout upon PostgreSQL installation, proves invaluable from two perspectives:

1. **Expansion:** When the volume or partition where the DB is created runs low on space, tablespaces can be created on different partitions or disks. This flexibility allows the DB to be expanded until the system can be reorganized
2. **Performance Optimization:** For instance, high-performance SSDs can be utilized for tablespaces to host frequently updated, high-usage indexes, thereby enhancing performance. This optimization technique can significantly improve the efficiency of database objects.

Conclusion: PostgreSQL as a Foundational Relational Database

Exploring PostgreSQL's architecture offers insight into a relational database designed for high efficiency and flexibility. From its shared memory and process management to the strategic use of tablespaces, PostgreSQL stands out as a comprehensive solution for modern data management challenges.

As a relational database, PostgreSQL is not just a tool for data storage but a foundation for scaling, optimizing, and transforming data landscapes.

Ready to see what PostgreSQL can do for your database needs?

Postgresql | Postgresql Database | Relational Databases | Postgresql Architecture | Open Source Software

Published in Bitnine Global

27 Followers · Last published Oct 18, 2024

Follow

Multi-Model Hybrid Graph Database Solution. Combines the strengths of Relational + Graph databases into one powerful solution. Visit Us : https://www.bitnineglobal.com

Written by Bitnine Global

46 Followers · 33 Following

Follow

Database Management Solution Provider Relational DB + Graph DB / Graph Data Modelling + Analysis + Visualization Visit Us: https://www.bitnineglobal.com

No responses yet

Write a response

What are your thoughts?

More from Bitnine Global and Bitnine Global

In Bitnine Global by Bitnine Global

Introduction to PostgreSQL Extensions

PostgreSQL Extensions: Supercharge Your Database

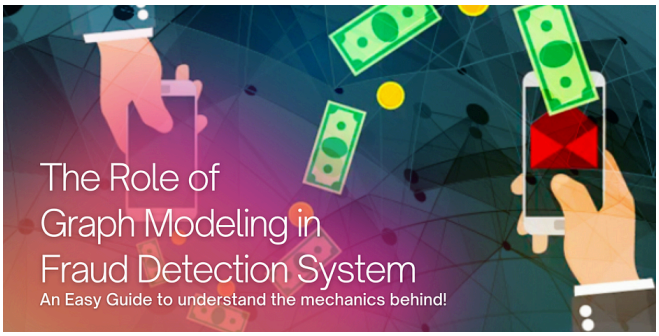
Jul 24, 2024 14 2

In Bitnine Global by Bitnine Global

Converting Oracle PL/SQL to PostgreSQL PL/pgSQL

Welcome to the world of database programming, where the languages of...

Jul 24, 2024 15 2



In Bitnine Global by AGEDB

The Role of Graph Modeling in Fraud Detection Systems–An eas...

Fraud poses a significant threat to organizations across various industries,...

Jun 7, 2023 27 1

In Bitnine Global by Bitnine Global

Key Features of each Version of PostgreSQL

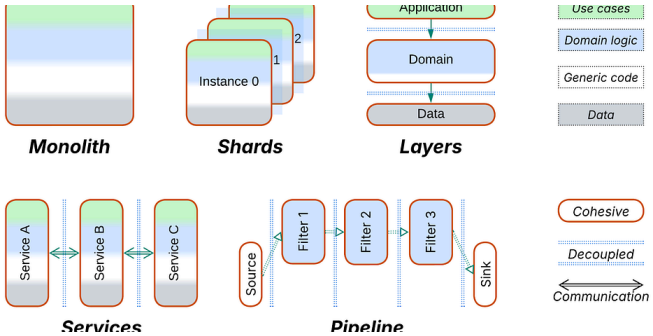
PostgreSQL, an open-source relational database management system, has been a...

Jan 25, 2024 3

See all from Bitnine Global

See all from Bitnine Global

Recommended from Medium

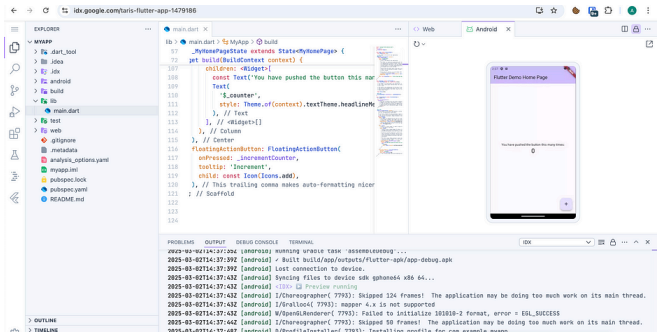


In ITNEXT by Denys Poltorak

Deconstructing patterns

The heart of software architecture, part 2

Apr 7 204 1

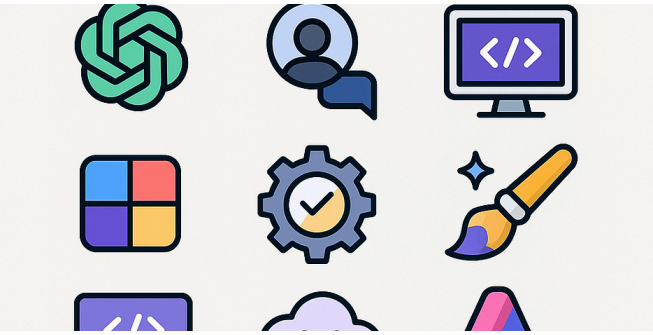


In Coding Beauty by Tari Ibaba

This new IDE from Google is an absolute game changer

This new IDE from Google is seriously revolutionary.

Mar 12 4.3K 246



In Let's Code Future by TheMindShift

10 AI Tools That Replace a Full Dev Team (Almost)

The future isn't coming—it's already here. And it's writing your code, fixing bugs, and...

Apr 6 1.3K 54

In Level Up Coding by Jacob Bennett

The 5 paid subscriptions I actually use in 2025 as a Staff Software...

Tools I use that are cheaper than Netflix

Jan 8 12.4K 315

In JavaGuides by Ramesh Fadatare

5 Microservices Design Patterns You Must Know in 2025

Here are five important microservices design patterns you should know in 2025, explained...

Jan 25 638 14

Harendra

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 26, 2024 9.8K 185

See more recommendations